# Property Probes

## Source Code Based Exploration of Program Analysis Results

Anton Risberg Alaküla
anton.risberg_alakula@cs.lth.se
Lund University
Lund, Sweden

Görel Hedin
gorel.hedin@cs.lth.se
Lund University
Lund, Sweden

Niklas Fors
niklas.fors@cs.lth.se
Lund University
Lund, Sweden

Adrian Pop
adrian.pop@liu.se
Linköping University
Linköping, Sweden

## Abstract

We present *property probes*, a mechanism for helping a developer interactively explore partial program analysis results in terms of the source program, and as the program is edited. A node locator data structure is introduced that maps between source code spans and program representation nodes, and that helps identify probed nodes in a robust way, after modifications to the source code. We have developed a client-server based tool supporting property probes, and argue that it is very helpful in debugging and understanding program analyses. We have evaluated our tool on several languages and analyses, including a full Java compiler and a tool for intraprocedural dataflow analysis. Our performance results show that the probe overhead is negligible even when analyzing large projects.

***CCS Concepts:*** • **Software and its engineering** → *Integrated and visual development environments*; *Compilers*; • **Theory of computation** → *Program analysis.*

***Keywords:*** program analysis, debugging, property probes

**ACM Reference Format:**
Anton Risberg Alaküla, Görel Hedin, Niklas Fors, and Adrian Pop. 2022. Property Probes: Source Code Based Exploration of Program Analysis Results. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3567512.3567525


*SLE '22, December 06–07, 2022, Auckland, New Zealand*

ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567525

## 1 Introduction

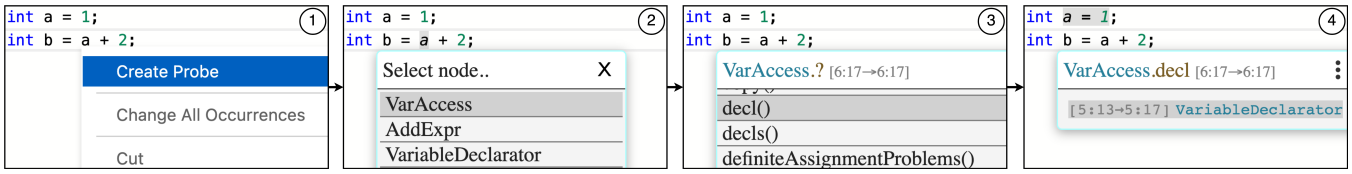
Modern software tooling includes many kinds of program analysis. For instance, compilers do type analysis, IDEs support type-based navigation and editing, and bug-finding tools may use analyses based on dataflow and effects. However, developing new analyses can be difficult. There are often many subanalyses, and they might need to handle many corner cases of the analyzed language.

In this paper, we propose a new interactive mechanism, *property probes*, to help the analysis developer. The main idea is to allow the developer to inspect and display *properties*, i.e., (partial) analysis results tied to specific parts of an editable source code (as plain text). The developer can interactively explore analyses by creating probes for different program elements, and the results are updated as the source code is edited, and even after updates to the analysis tool itself.

It is a challenge to match a probe for a particular property to the corresponding program element after source code mutations, and we provide a robust algorithm for this purpose. Our approach also supports the probing of properties of implicit program elements that are not directly visible in the edited source code, e.g., imported libraries or predefined elements built into the programming language, like `class Object` in Java. We see property probes as a complement to traditional development support such as automated tests and traditional breakpoint/step-debuggers or "print debugging".

We have implemented a property probe tool, CodeProber, specifically targeting analyses built with Reference Attribute Grammars (RAGs) [12]. The attributes of an attribute grammar match the probed properties, and the interactive probing fits the demand evaluation used in RAGs. However, the concept of property probes can in principle be applied to any analysis that uses an abstract syntax tree (AST) as the spanning tree over its program representation, and that associates partial analysis results with nodes of the tree. We therefore expect the ideas to be useful for analyses built with a much wider range of approaches than RAGs.

We have applied CodeProber to a number of different languages and analyses, in particular to ExtendJ [7], a full




(1) The user first right-clicks on the use of variable a in the source text and selects "Create Probe".

(2) Then, the AST nodes that span the clicked variable appear in a menu, and the user selects the desired node, `VarAccess` in this case.

(3) The user is presented with a menu of all available properties on the `VarAccess` node, and selects the `decl` property.

(4) Finally, the window for the property probe appears, showing the result value, in this case a reference to the variable declaration, represented by a `VariableDeclarator` node.

**Figure 1.** Interactively creating a property probe

Java compiler, and to IntraJ [16], an extension of ExtendJ that supports intraprocedural control-flow and dataflow analysis.

Our contributions are as follows:

- We introduce the concept of property probes, and illustrate typical usage of them in CodeProber (Section 2).
- We present an algorithm for robust identification of the probed AST nodes after mutations of the source code. We also present known limitations of the algorithm (Section 3).
- We present the architecture of CodeProber, and the requirements an AST must follow to be used with CodeProber (Section 4).
- We present experiences from using CodeProber in case studies, and performance measurements to explore its limitations with regards to the size of the edited code and number of active probes (Section 5).

Finally, we present related work in Section 6, and then conclude in Section 7.

## 2 Property Probes

In this section, we present the concept of a property probe, and the overall architecture of a property probe tool, using CodeProber for examples.

### 2.1 Property probes

A property probe is an interactive element, presented in the context of a source code text editor, and acting as a live observer of a property of an AST node. In CodeProber, each property probe is displayed as a small window.

Internally, a probe is represented by a *node locator* (a way of identifying a particular node in the AST), a *property name*, optionally a number of *arguments* (if the property takes arguments), and a *result value*, i.e., the most recently computed value of the property. The result value is a collection of primitive values, like string or integer, and AST node references (represented as node locators).

A probe has two main responsibilities:

1. It adjusts the node locator after source code edits.
2. It presents up-to-date results to the user, reevaluating the property as needed.

Evaluating a property means, in practical terms, invoking a function in the context of an AST node. Keeping the result up-to-date means re-invoking the same function whenever needed, such as when the source code is modified.

The user can create a probe starting from a location in the text, selecting the desired AST node in case several nodes match the same location. It is also possible for a user to create a new probe starting from the result of another probe. This allows property exploration not only directly related to the edited text, but also by exploring probe results, which can again be explored further, supporting an interactive way of investigating partial results of an analysis.

Exploration of probe results opens for exploring properties of nodes that have no matching location in the edited source text. One example is nodes corresponding to ASTs of imported libraries. Another example is synthetic nodes created to represent implicit program entities. This could be implicit types, like `class Object` in Java, or desugared representations of language constructs, or computed complex properties resulting from some analysis. Attribute grammars can use higher-order attributes for such purposes, i.e., attributes whose values are new AST nodes that may themselves have attributes [20].

### 2.2 Examples

Figure 1 shows an example of using CodeProber on the ExtendJ Java compiler to create a probe.

The user right clicks in the source code and selects the menu option "Create Probe" (1). A list appears, showing AST nodes that overlap with the clicked location. The user selects one of them (2). A new list appears, showing all properties available on the selected AST node. The user selects a property (3). A window (property probe) appears which shows the result of evaluating the selected property on the selected node (4). The user can then, if desired, use the probe to bring up more probes. This can be done either by investigating more properties of the same node by clicking on the title of the probe, or by investigating properties of the result node, by clicking on the result value.

It can be noted that for node locators that correspond to a text location, CodeProber shows the text location in the probe, both for probed nodes and for results. These locations

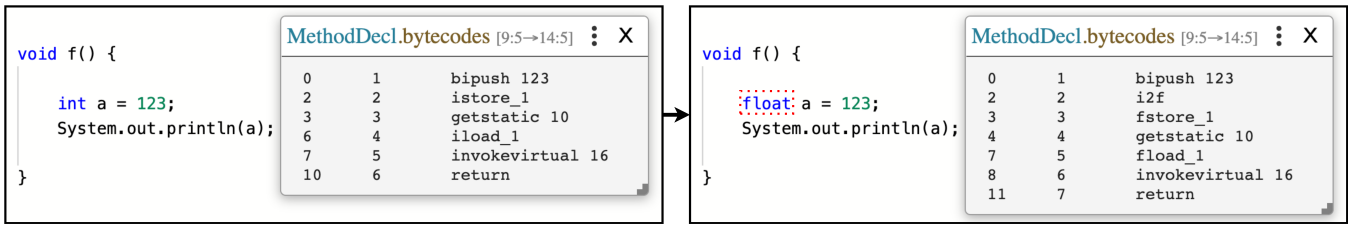



**Figure 2.** Probe result automatically updated after changing type from `int` to `float` (dotted box).


**Figure 3.** Probe on a synthetic Exit node.

are hoverable, making the corresponding text fragment light up. In the example, the user hovers over the probe result, making the variable declaration in the source text light up.

Figure 2 shows an example of how a probe is updated when the user edits the source code. In this case, the user has created a probe for the bytecode of a method. When the user edits the source code, changing the type of the variable from `int` to `float`, the bytecode in the probe result is updated, for example, changing the `istore_1` and `iload_1` instructions to `fstore_1` and `fload_1`.

Figure 3 shows an example of a probe on a synthetic node. Here, CodeProber is run on IntraJ, an analysis tool that adds intraprocedural control flow and dataflow to the ExtendJ Java compiler. When IntraJ constructs the intraprocedural control-flow graph for a method it also creates synthetic `Entry` and `Exit` nodes, and gives them positions based on the start/end of the method. The probe in the figure is evaluated on the `Exit` node for `f`, and the probe result shows the predecessors for `Exit`, i.e. last statements before method `f` ends. The user hovers over the location in the title row of the probe, and the corresponding curly brace on line 12 is highlighted.

### 2.3 Tool Architecture

To implement property probes for a specific analysis tool, we propose a client-server architecture, see Figure 4. The client side uses a customizable code editor such as Monaco [4] or similar. The server side consists of two components; a server and an analysis tool.

It is the responsibility of the analysis tool to parse the edited text into an AST (as well as any other files needed for the analysis), and to populate the AST with the functionality to be explored using probes. The server uses an API to access the analysis tool, e.g., to get the root of the current AST and to query properties on different AST nodes. The server also handles the communication with the client-side code editor.

The probes are stored on the client-side, using node locators as references to node objects. This allows the AST to be reparsed at any time, or even the analysis tool to be restarted from scratch during the editing session. This also allows the server to be completely stateless.

In CodeProber, the client is a web page, mostly written in TypeScript, and using the Monaco code editor. The server is written in Java, and can use any analysis tool written using the JastAdd metacompiler (packaged as a jar file, following certain conventions). For example, in Figures 1 and 2, the underlying analysis tool is ExtendJ [7], a Java compiler implemented using JastAdd, whereas in Figure 3, the analysis tool is IntraJ [16], an extension to ExtendJ that adds intraprocedural analysis. In principle, other kinds of analysis tools could also be used if they are adapted to follow the same conventions as JastAdd (discussed more in Section 4).

For practical purposes, CodeProber packages the client and the server together as a single jar file which takes a path to the analysis tool as its argument. When started, CodeProber opens a local HTTP server that serves the webpage, and a local WebSocket server for all dynamic requests. The user can then simply go to the webpage and start editing and creating probes.

## 3 Node locators

As mentioned earlier, so called *node locators* are used to identify the AST nodes referenced by probes. These node locators need to be updated after mutations of the source code in the text editor. There are many potential ways to identify where an AST node is located. Some examples in plain English are:

1. "The call expression on line 12, column 9"
2. "The third child of the fifth child of the root AST node"
3. "The class declaration with ID set to 'Foo'"

These ways of identifying nodes might work, but they are fragile to changes: The first example will break if, for example, a statement is added at the beginning of the source code; The second example will break if, for example, the construct of interest is nested inside a new statement; The third example will break if, for example, the class is renamed to "'Bar'".

Furthermore, there can be probes on synthetic nodes that have no textual representation, and that require more sophisticated identification methods.

We have designed the node locators with the goal of making them both resilient to different kinds of mutations of the source code, and efficient to apply, i.e., to resolve them to actual object references. Another design goal is that node locators should be as language agnostic as possible, and not make assumptions of how source text is parsed. This prevents potential solutions that rely on inserting tracking markers in the code, for example using annotations or block comments, as annotations and block comments aren't supported in all languages. In addition, such tracking markers would not be possible to use with synthetic nodes.

Our current design is the result of an iterative development process where we have tried out probes on many different properties for several different analysis tools and for different languages. In our experience the design works very well in practice, although there will always be cases when node locators can fail, for example when the corresponding code is completely removed.

### 3.1 Node Locator Steps

A node locator is a list of *steps* where each step moves a current position to a new position in the AST. The steps are applied in order, starting at the root of the AST. Any of the steps can fail, in which case the whole application of the node locator fails, and no node could be identified. The following steps are supported:

**Child** has the form
`Child(`$i$`)`
It means go to the $i$'th child node.

**TAL** stands for *Type At Location* and has the form
`TAL(`$t, d, l_s : c_s \rightarrow l_e : c_e$`)`
Here, $t$ is an AST node type, $d$ is a number of steps down in the AST, and $l_s : c_s \rightarrow l_e : c_e$ is a line/column start/end span in the text. This step moves the current position to the "best" node of type $t$ in the subtree of the current position and whose text span has at least one character overlap with $l_s : c_s \rightarrow l_e : c_e$.[1] "Best" here means the node closest to being at depth $d$ from the current position. In case there are several such nodes, then the node whose start/end most closely matches the TAL start/end is picked. In case there are still multiple identical matches, the first one in a depth-first traversal is chosen.

**FN** stands for *Function* and has the form
`FN(`$f$`, `$a_1$`, .., `$a_n$`)`
where $f$ is the name of a function on the current node, and $a_1, .., a_n$ are arguments to the function ($n \geq 0$). The function is expected to return an AST node reference, and the current position is moved to that node.

The Child steps provide a simple way of locating a node in an AST, but it is not very resilient to changes. Even a small change, like changing something in the beginning of the source code, would result in old node locators to fail or resolve to the wrong node in the new AST.

The TAL steps are introduced to provide a resilient solution. They can handle variations in the placement of the nodes due to additions, deletions, and nesting changes. The TAL steps also make use of text spans in the edited source code. For this to work, the editor adapts the TAL text spans in its stored probes as the text is edited. If, for example, the user inserts a newline between lines $N$ and $M$, then for all TAL steps on line $L \geq M$, the line count must be increased by 1. Similar adjustments must be made for lines and columns on all insertions and removals.

The FN steps were introduced to handle synthetic nodes, constructed by the analysis tool in a different stage than parsing. In particular, they can be used for higher-order attributes (HOAs), in which case the function is simply the name of the HOA, and returns the root of the HOA subtree. In Reference Attribute Grammars, the HOAs are evaluated on demand and memoized, so in case the attribute had not already been accessed for other reasons, calling the function will result in the HOA being created before its root is returned.

The FN steps might be useful also for other purposes. They are very versatile as the function may return any node in the AST. One possible use might be to introduce application-specific steps, such as jumping to a particular declaration node. However, we have not experimented with that, since our main goal has been to provide an algorithm that works out of the box for any language and analysis tool.

### 3.2 Example Node Locators

In the Java compiler ExtendJ, the root AST node is of type `Program`. `Program` has a `List`, which in turn contains a number of `CompilationUnit` nodes, each corresponding to a single source file. Most AST nodes for a source file can be identified by first identifying a `CompilationUnit`, and then using a TAL within that file.

As an example, assume we have the following variable declaration at line 5 in a given source file:

```
int a = 1;
```

An example node locator for the variable declarator (`"a = 1"`) is:

```
[ Child(0),
  Child(131),
  TAL("VariableDeclarator", 10, 5:13 -> 5:17) ]
```

Here, `Child(0)` goes from the root AST node (`Program`) to the `List` node. `Child(131)` goes to the 132:nd `CompilationUnit`, which happens to represent the source file. "10" is the number of steps from the `CompilationUnit` down to the

[1] Nodes that are not given explicit spans by a parser should have the span $0:0 \rightarrow 0:0$ and are considered to overlap any other span.

`VariableDeclarator` node. "5:13" is the starting line and column and "5:17" is the ending line and column.

For library compilation units, we rely on FN instead, since libraries are implemented using higher-order attributes in ExtendJ.

For example, to identify the `Integer` class, we would use:

```
[ FN("getLibCompilationUnit",
     "java.lang.Integer"),
  TAL("ClassDecl", 2, 0:0 -> 0:0) ]
```

Here, the FN step represents the function call `getLibCompilationUnit("java.lang.Integer")` on the root node, and results in a `CompilationUnit` node. The TAL step then locates the `ClassDecl` two steps down from the `CompilationUnit`. The text span in this case is `0:0 -> 0:0`, which is expected for AST nodes that do not get created from a normal source file.

### 3.3 Creating Node Locators

To create a node locator for an AST node $n$, the following two stages are performed:

***Create*** Create a list of steps corresponding to the path from the root down to $n$. For each edge on the path, the corresponding step is either a Child (for a normal child), or an FN (for a higher-order attribute).

***Merge*** Merge sequences of 1 or more Child steps into a single TAL step if the TAL step is unambiguous, i.e., if applying it results in exactly one node.

Strictly speaking, only `Create` is necessary. `Merge` exists to reduce the risk of failure when applying the locator later, after changes to the code.

As a simple example of when a TAL can be ambiguous, consider the following statement:

```
i += 1;
```

In some languages this is equivalent to writing:

```
i = i + 1;
```

Suppose the analysis tool transforms the AST by replacing the first statement by the second one, in order to obtain a normalized AST that is easier to analyze. A question then is what text location information to associate with the nodes in the transformed statement. One possibility would be to not set the location at all (i.e., using the default `0:0 -> 0:0`). Another possibility is to use the text location from `"i"` in the original statement for both occurrences of `"i"` in the transformed statement. Either way, it is likely that the two variable reference nodes get identical location and type. This would make a TAL locator ambiguous, hence the need to take ambiguity into account when performing the `Merge` step.

### 3.4 Adapting to Mutations

As mentioned, the client keeps track of the active probes with their node locators, and adjusts the text spans of the TAL steps as the code is edited. However, the client does not have any knowledge of the syntax, so it only adjusts according to textual changes. The consequence is that the node locator might not fit exactly with the AST of the reparsed code. The flexibility of the TAL step usually makes it possible to find the node, but this also means that there can be a better, more exact node locator that should be used in the future. For this reason, when the server sends updated probe results to the client, it also sends the updated node locator for the probe.

As an example, assume we have the following source code:

```
if (x) {
    a();
}
if (x) {
    b();
}
```

The node locator for the first if-statement contains a TAL step `TAL("IfStatement", 3, 1:1 -> 3:1)`. Suppose now that the user decides to clean up some duplicated code, so they remove the two lines in the middle. The source code now looks like the following:

```
if (x) {
    a();
    b();
}
```

Because of the removed lines, the client adjusts the TAL step to `TAL("IfStatement", 3, 1:1 -> 2:9)`, covering only the first two lines (up to and including the `a()` statement). The client then informs the server that there are changes, and the server then sends back the updated probe result, together with a new more appropriate node locator with a `TAL("IfStatement", 3, 1:1 -> 4:1)`.

### 3.5 Known Limitations

The proposed design for node locators has some known limitations. This section describes those limitations, and some potential ways to solve them.

**3.5.1 False Positives.** Node locators can sometimes give false positives. For example, assume we have the following expression:

```
a(b(c))
```

The user adds a probe for `b(c)`. The locator for that probe looks like this:

```
[ TAL("CallExpr", 7, 5:13 -> 5:16) ]
```

Then the user changes the source code to:

```
a(c)
```

After the change, the probe will be re-evaluated on `a(c)`, since it is the only AST node that matches `CallExpr` and overlaps with the original TAL position. If the user wanted the probe to keep matching the first argument to `a`, then

matching `a(c)` is a false positive. The user would rather match `c`.

One potential solution to this scenario is to make use of subtyping: The call `b(c)` has the AST node type `CallExpr`, and according to the abstract grammar, its parent expects any node of the supertype `Expr` at that position. By using `Expr` instead of `CallExpr` in the TAL step, the new expression `c` would match, solving the user's problem.

Node locators need to balance resilience and risk of false positives when deciding how strictly they should match nodes. We found that being strict with types and permissive with locations seems to work well. Being less strict with types (for example by using subtyping) could potentially introduce more false positives, even if it would help the specific example above.

Perhaps locators could search both with exact types and supertypes simultaneously, and use some heuristic to sort the results. Building UI/UX that supports this in an understandable way is an interesting challenge.

#### 3.5.2 Ambiguous Intent When Creating Locators.

When a property result is a node reference, the user can click on the reference to create new probes based on the resulting node. The meaning of that click can be interpreted in two different ways. Did the user want to create a probe for..

- ..the node they clicked on?
- ..the result of the property?

In more concrete terms, if the user clicks on the `VariableDeclarator` in step 4 in Figure 1, then we can identify it in one of the following ways:

```
[ TAL("VariableDeclarator", ..) ]
[ TAL("VarAccess", ..), FN("decl") ]
```

We use the first option, i.e. we don't keep any reference to the original `VarAccess` node. But there is an argument to be made for using the longer locator instead, as it would allow the user to create more complex probes. However, there are a few challenges to solve:

- By using a longer locator, performance is expected to degrade. The more steps, the more time it will take to create and apply the locator.
- Building UI/UX that explains longer locators is a challenge.

## 4 Implementation

The overall architecture of CodeProber has three components: a client, a server, and an analysis tool, see Figure 4. The implementation is open source and available on https://github.com/lu-cs-sde/codeprober. This section describes the APIs between the three components, together with rules and recommendations on how the AST produced by the analysis tool should work.

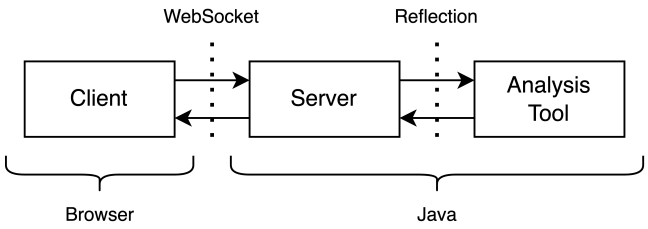



**Figure 4.** High level architecture

### 4.1 Client↔Server API

The client and server communicate with remote procedure calls (RPC) over WebSocket. There are three different request types sent by the client: ListNodes, ListProperties and EvaluateProperty, corresponding to the three steps when the user creates a probe, as in Figure 1. The client sends the EvaluateProperty request also when the user edits the text, as in Figure 2.

There is one message sent from the server to the client; Refresh. This is sent when the server detects that the underlying analysis tool has been updated. The client is not expected to respond to this message, but will instead re-evaluate all active probes by sending EvaluateProperty requests.

In all requests sent from the client to the server, the client includes the current editor state, i.e., the full text. The server will then ask the underlying analysis tool to parse this text into a new AST. For responses to requests on node locators, the server includes updated node locators in the response, based on the new AST. The client then uses the new locator in subsequent requests. This way, the node locators on the client side are constantly adapted to the most recent AST, as was discussed in Section 3.

**Example.** Figure 5 shows a sequence diagram for the messages sent when the user creates the probe in Figure 1. In the first step (1 → 2), the user clicks in the text to create a probe. The client then sends the ListNodes request, with the full text and the cursor position as arguments. In response, the server sends a list of node locators, corresponding to the nodes that match the cursor position.

In the second step (2 → 3), the user selects one of the nodes. The client then sends the request ListProperties, with the full text and the node locator as arguments. In response, the server sends an updated node locator, along with a list of property identifiers, each containing a property name and its argument types.

In the third step (3 → 4), the user selects one of the properties. The client then sends the request EvaluateProperty, with the full text, the node locator, the property identifier and any arguments to the property. (If the property has arguments, the client prompts the user to supply them interactively.) In response, the server sends the updated node

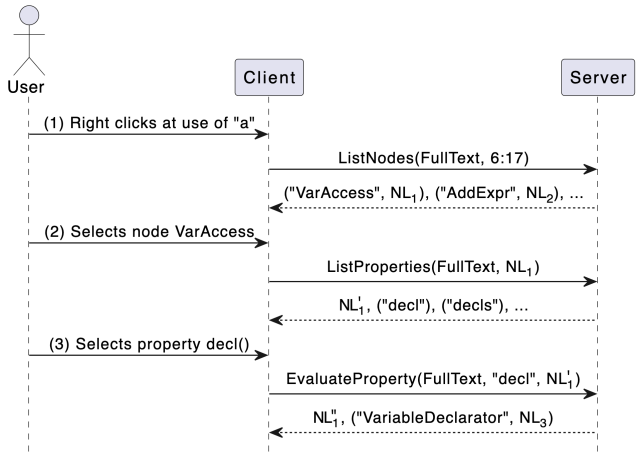



**Figure 5.** Sequence diagram for Figure 1

locator, as well as the probe result. Any AST nodes in the result are encoded as node locators that can be used for future ListProperties requests.

For the scenario in Figure 2, the client will send one EvaluateProperty request for each of the active probes.

### 4.2 Server↔Analysis Tool API

The concept of property probes can in theory be used for analysis tools implemented in any language, but CodeProber is written in Java and currently requires the analysis tool to run on the JVM. The server communicates with the analysis tool using reflective calls.

The AST is parsed or reparsed by calling the `main` method on the analysis tool jar file, with the path of a temporary file containing the client state (the full text). The main method should store the parsed AST in a static field in the main class, that should be declared as follows:

```
public static Object DrAST_root_node;
```

This integration strategy is also used by DrAST [13], another AST exploration tool. For simplicity, we use the same field name, since many tools built with the JastAdd metacompiler are already prepared to work with DrAST.

Once an AST is produced, the server uses reflection to access and traverse it. The server assumes that the AST follows a certain structure. A number of methods should be available on each AST node for traversal purposes:

1. **getNumChild()** - returns the number of children on this node.
2. **getChild(int)** - returns a child at a given index.
3. **getParent()** - returns the parent node for this AST node, or *null* for the root AST node.
4. **getStart() / getEnd()** - returns the start/end position for this AST node.

When listing available properties, reflection is again used, via *java.lang.Class.getMethods()*. The full list is filtered before being returned to the client. Methods that are non-public or contain "_" or "$" in the name are removed. Methods with arguments can only contain arguments of type `int`, `boolean`, `String` or AST node references. Methods with other argument types are removed.

To compute node locators, it must be possible to determine the connection between a parent and child AST node in the form of either a Child or FN step.

Child steps are determined by iterating over all children in the parent node. Using identity comparison, we can detect the index of the child node.

If a node is a higher-order attribute (HOA), an FN step should be used, including the name and the arguments of the HOA. This can be determined thanks to the fact that JastAdd caches the arguments to, and results of, all HOA invocations. To construct an FN step for a HOA, CodeProber selects the parent of the HOA, and then iterates through all the parent's HOA caches, again using identity comparison to find the name and arguments of the appropriate child HOA.

In case no parent/child connection can be determined, the child node is considered to not be attached to the AST. This causes node locator creations to fail, and the server sends an error code to the client.

### 4.3 Desirable AST Features

To use property probes, some design choices for the analysis tool are desirable to help improving the user experience: good source locations in the AST, on-demand evaluation of properties, and pure properties. We will discuss these in turn.

**Source locations.** For property probes to work well, it is desirable that the parser captures line and column positions and stores them in the AST nodes at parsing. Also, the positions should honor the AST hierarchy: CodeProber assumes that a node with an explicitly set position has an equal or larger span than all nodes in its subtree. Otherwise, our TAL algorithm might miss the best matching node, since it uses positions to prune subtrees in its search.

In many tools, nodes do get appropriate positions, in order for the tool to be able to report locations of errors and warnings. However, in a given tool, there might be nodes that lack this information. Perhaps locations have been added only for the nodes with associated error messages; Perhaps the AST is transformed, and location information is not carried over to the transformed parts.

Missing locations will degrade the user experience, as features like highlighting and right clicking to select AST nodes will not work. In the client of CodeProber, a small warning triangle is shown next to each AST location that has its line and column set to zero.

However, CodeProber also tries to compensate for missing location information. It uses a *position recovery strategy* to infer suitable location information in case it is missing. There are multiple supported strategies, and the user can

select which (if any) to use. The default strategy looks at nearby parent and child nodes, progressively searching further up and down in the AST until a valid location is found. A recovered position usually covers a slightly larger or smaller span than the real span of the AST node. Therefore, position recovery should be seen as a temporary solution, and it is better if all AST nodes carry their own position instead.

**On-demand evaluation.** The user can create probes for any property the AST supports, but usually only a tiny subset of the functionality is ever probed for at the same time. This fits well with on-demand evaluation. Rather than computing everything up front, it is advantageous if properties are lazy and their values computed only when demanded. On-demand computation is not strictly necessary, but if all potentially probed values are computed up-front, as soon as the source text is edited and the AST is reparsed, re-evaluation of all these values might take a long time and be at odds with the user experience. Of course, if the user does not edit the source text, but only explores properties, the probes can still be very valuable for tools that do up-front computations.

**Pure properties.** The probed properties should be observationally pure, i.e., without visible side-effects when accessing them. If accessing properties has side-effects, and they behave differently depending on in which order they are invoked, then the benefits of property probes diminishes. In addition, there is a caching setting in CodeProber that greatly improves performance by reusing the AST whenever possible, to avoid unnecessary reparsing. If properties can cause mutations in the AST, then caching is not reliable.

All our evaluations have been performed with JastAdd-based tools, where all property evaluation is on-demand and all properties (attributes) are observationally pure.

### 4.4 Program Representation

In this paper, we have assumed that the program representation is an AST. However, it is sufficient if the representation has a spanning tree, with the traversal interface discussed above, and where source text locations can be attached to the nodes in the spanning tree. In fact, this is the case for JastAdd tools: Many of the JastAdd attributes are node references, so the program representation is actually a graph, but with the AST as the spanning tree.

Many analysis tools work on the bytecode level, and perhaps use a control-flow graph as their main representation. We have not experimented with using property probes for such tools, but we think it should be possible to use a suitable spanning tree consisting of hierarchies of, for example, files, classes, methods, basic blocks, and instructions, all annotated with suitable source code locations.

## 5 Evaluation

This section presents the findings from case studies and performance measurements of CodeProber.

### 5.1 Case Study: ExtendJ

During development of CodeProber we continuously tested functionality against the Java compiler ExtendJ. Based on this experience, we added a few features that we think are useful also for many other analysis tools.

The *position recovery strategy* mentioned in Section 4.3 was added specifically because some node types in ExtendJ don't carry their own position information.

We also added a few features that improve multi-file support. We noticed that when we added multiple external files to ExtendJ, some problems appeared. For example, the performance of creating and applying node locators initially scaled poorly with the number of external files. The issues stemmed from CodeProber traversing through the entire AST, when it should ideally have avoided AST nodes that correspond to external files. We added support for a few optional functions that can tell CodeProber to not traverse through certain nodes, which is used in ExtendJ to avoid unnecessarily traversing through external files. This greatly improves the performance of creating and applying node locators.

During development we also identified and fixed two different caching issues in ExtendJ, which we contributed back to ExtendJ. We hypothesize that these issues hadn't been discovered before because ExtendJ had not before been used in the live, incremental way that CodeProber uses its underlying analysis tools.

In general, our experience is that using property probes with ExtendJ has been an excellent way of building understanding of the compiler. One common use case for ExtendJ is to write program analysers or experimental extensions to Java. To do this you need to know what AST node types are available, and what properties they contain. This can be accomplished by consulting the official API documentation [3]. However, as CodeProber took shape, we found ourselves using it more often instead. For example, if you want to know what properties are available on a "for-each" statement, then you can write such a statement, right click on it and see the list of properties. If any property looks interesting, you can click it to immediately see how it works. With the API documentation you need to first find the name of the node ("EnhancedForStmt") and then you get a list of property names, but these cannot be directly invoked since there is no concrete code attached.

### 5.2 Case Study: IntraJ

IntraJ [16] is an extension to ExtendJ that adds intraprocedural control-flow and dataflow analysis. The main author and developer of IntraJ used CodeProber in developing new types of analysis. Before using CodeProber, the IntraJ developer had used the following cycle when developing new features:

- Write code for the new feature.

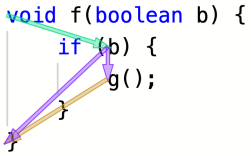



**Figure 6.** Control-flow graph rendered on top of code.

- Add print statement(s) to check that the feature works correctly ("print debugging").
- Iteratively modify the code and print statements until you get the expected behavior.
- Remove the print statement(s).

Now, property probes have replaced most of the "print debugging" steps, since it is much faster and simpler to open/-close probes than it is to add/remove print statements and recompile IntraJ.

The IntraJ developer also mentioned that they use the ExtendJ API documentation less, since it often is quicker to explore functionality via the property probes in CodeProber.

One new feature was added specifically for IntraJ; the possibility to visualize the control-flow graph directly in the source code. Previously, the control-flow graph was usually inspected in textual form, which was inconvenient, or on a dot visualization of the AST with control-flow edges, but it could become very large even for a small program. We therefore added a feature that allows property probes to declare arrows to be drawn between two positions in the source code, and used this to display the control-flow graph, as seen in Figure 6. The implementation was inspired by the bug explanations in Clang Static analyzer [2].

Each probe is responsible for zero or more arrows. When you close a probe, its associated arrows disappear too. This means that the amount of currently visible arrows depends on which probes are open. The user can choose to show the full control-flow graph for the entire program, or just for a single method, etc.

### 5.3 Case Study: Other Compilers

We used CodeProber with three different compilers for the languages Oberon-0 [9], Bloqqi [10] and SimpliC, a simple C-like language. The experience worked well for all compilers. We did, however, discover that every compiler had a few AST nodes that didn't carry correct location information. Nodes that produced errors/warnings generally had correct locations. Missing locations were usually attributed to either desugaring or that multiple nodes were created in the same parser production (the parser generator attached location only to the return node of a production). This was no major issue however since the *position recovery strategy* developed for ExtendJ worked fine here, as well.

We have a compiler course where students create SimpliC compilers during the lab sessions. We presented CodeProber to ~70 students early on in the course, and later asked about their experiences using it. We found that many students thought CodeProber was useful and they kept using it throughout the entire course. They used it to explore and debug name and type analysis, call graph construction, code generation, and more. Based on the feedback we received, CodeProber will be used in future iterations of the course as well.

We also used CodeProber with a student implementation of a ChocoPy [15] compiler. ChocoPy is a subset of Python, commonly used for educational purposes. Here we had more severe location-related issues. One of the challenges with parsing Python is the indentation sensitivity. The ChocoPy implementation we used had solved this by making the parser a two-step process; first it transforms all indentation into whitespace-insensitive indentation tokens, and then it parses the transformed source code. AST nodes produced from the transformed sources always had line numbers that matched the original source code, but their columns were usually wrong by a few characters, since the whitespace-insensitive tokens didn't match the width of the original indentation. You could still create probes, but the user experience was not very good. Of course, these problems could easily have been solved by fixing the ChocoPy implementation to set proper line and column numbers, but it illustrates the kinds of problems a user can run into.

### 5.4 Performance

We have measured the performance of property probe-related operations in CodeProber in order to find out the limitations on sizes of projects and number of active probes. This section presents the methodology behind the measurements, and the results.

The processing time it takes to use property probes can be divided into three main groups:

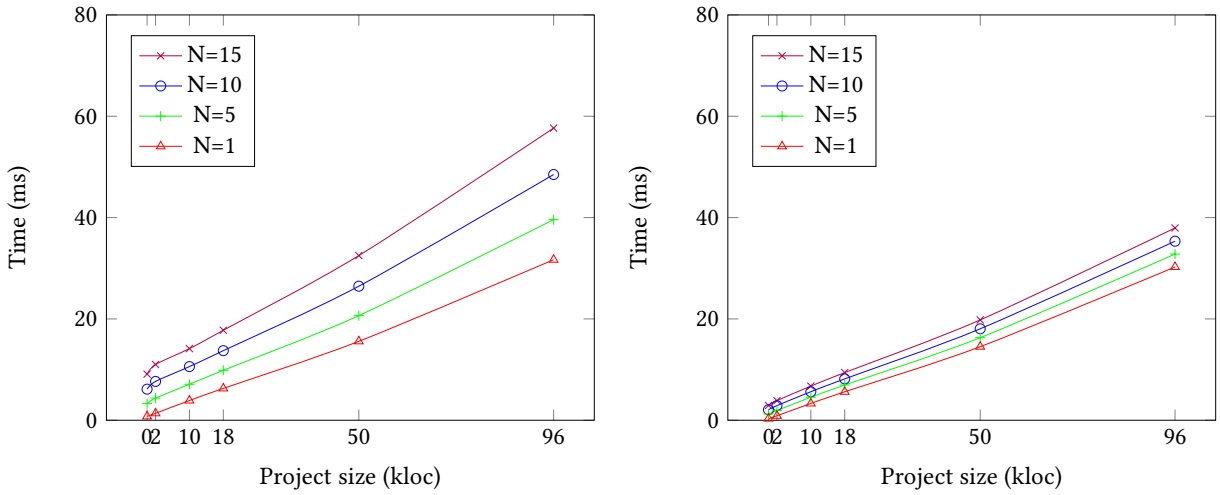
- Parsing
- Property evaluation
- Probe administration

Parsing and property evaluation can be mitigated to some extent, for example by doing incremental parsing or keeping fewer probes active. The processing time for these groups depend on the underlying analysis tool, which is ExtendJ in our benchmarks. ExtendJ has support for incremental parsing on a file level. I.e., when the user changes a file, we only need to re-parse that file, and then we can merge the result with all previously parsed files.

"Probe administration" contains all the functionality that is required to support property probes. This includes listing nodes that overlap with the user's cursor, creating and applying node locators, serializing probe results, etc.

The cost of parsing and property evaluation has to be paid even when ordinary debugging approaches are used, like unit tests or "print debugging". The administration cost is

**Figure 7.** Time to create and evaluate probes




**(a)** Create **N** probes

**(b)** Evaluate **N** probes

unique to property probes, and therefore we want to measure it to ensure that the overhead of using property probes is acceptable.

CodeProber is implemented with a client/server architecture, but all time-consuming work happens on the server side, so that is what the benchmarks are focused on. A headless client is used for measurements, and it runs on the same machine as the server in order to avoid any network latency in the data.

The measurements in this section were generated on a benchmark machine running Java 11 on Ubuntu 21.20 with an Intel i7-11700K CPU and 128GB DDR4 RAM. The machine is configured with a minimal amount of background services to reduce noise in the data. We also ran the same benchmarks on a normal development laptop, for comparison purposes. The laptop is running Java 17 on Mac OS 12.2.1, an Apple M1 Pro CPU and 16GB LPDDR5 RAM. The laptop results are not included in this paper, but they were roughly 1.5 times slower than the results from the benchmark machine.

Benchmarking is done with three variables: project configuration **P**, action type **T**, and number of actions **N**.

**P** is one of 6 project configurations. The minimum configuration is a single file containing 5 lines of code. The largest configuration is the source code of Apache FOP [1], which contain over 900 files and 96K lines of code.

The action type **T** is either creating or evaluating probes.

The number of actions **N** is 1, 5, 10 or 15. Whenever CodeProber performs more than one action for the same source code, it will reuse the AST. Therefore, the parsing cost only needs to be paid once per source code version. The overhead for action$_k$ ($k > 1$) is property evaluation time and probe administration.

For each combination of **P**, **T**, and **N**, we performed the following sequence:

1. Simulate a change to the source code.
2. Perform **N** actions of type **T**.

This sequence was performed in a loop until steady state had been achieved. After that we performed the sequence an additional 5000 times, and recorded the average time to finish the **N** actions in Table 1.

#### 5.4.1 Creating a Probe.

This involves two operations:

1. List all AST nodes overlapping with the user's cursor.
2. List all properties available on a given AST node.

The two operations correspond to the ListNodes and ListProperties requests in Section 4.1. Note that we don't measure EvaluateProperty. The result of the actions we simulate here is a correctly configured probe, but its result area in the probe will be empty until an EvaluateProperty request finishes, which we have benchmarked separately.

The first operation is skipped when a probe is created based on the result of an existing probe, for example by clicking on `VariableDeclarator` in step 4 of Figure 1. In our benchmarks we always measure both operations, which gives us a worst case time for creating a probe. The average probe creation time is expected to be slightly better, since you sometimes skip the first operation. The results can be seen in Figure 7a.

#### 5.4.2 Evaluating a Probe.

A probe is evaluated for one of two reasons:

- either a new probe was created,
- or some underlying data changed, and existing probes must be re-evaluated

**Table 1.** Performance measurements, all times in millisecond.
The data for creating and evaluating probes is plotted in Figure 7.

| Project Name | Size | Time to create **N** probes | | | | Time to evaluate **N** probes | | | | Full parse | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **LOC** | **N=1** | **N=5** | **N=10** | **N=15** | **N=1** | **N=5** | **N=10** | **N=15** | **Steady state** | **Startup** |
| Mini | 5 | 0.8 | 3.3 | 6.1 | 9.1 | 0.3 | 1.2 | 2.1 | 2.9 | 0.1 | 126.5 |
| Probe Server | 2K | 1.4 | 4.4 | 7.7 | 11.0 | 0.9 | 1.9 | 2.9 | 3.9 | 9.1 | 200.6 |
| Commons Codec | 10K | 3.9 | 7.1 | 10.6 | 14.1 | 3.3 | 4.5 | 5.6 | 6.7 | 44.0 | 287.6 |
| NetBeans | 18K | 6.3 | 9.9 | 13.7 | 17.8 | 5.6 | 7.0 | 8.2 | 9.4 | 58.1 | 346.8 |
| PMD | 50K | 15.6 | 20.6 | 26.5 | 32.5 | 14.5 | 16.3 | 18.1 | 19.8 | 153.9 | 492.3 |
| FOP | 96K | 31.7 | 39.6 | 48.5 | 57.7 | 30.2 | 32.8 | 35.3 | 38.0 | 358.0 | 720.1 |

Evaluating a probe corresponds to the EvaluateProperty request in Section 4.1. Probes are expected to be evaluated significantly more often than they are created, but exactly how often that happens depends on how you use CodeProber.

The property you evaluate has a large impact on the total evaluation time. We are mostly interested in measuring the probe administration time, so therefore we intentionally selected lightweight properties that finish in a short, constant time. Our measurements should therefore almost entirely consist of parsing time and probe administration time.

The results can be seen in Figure 7b.

#### 5.4.3 Full Parse Time.

The performance numbers shown in Figures 7a and 7b are using incremental parsing. For each measurement, all files in the project configuration are parsed once, and then only one file is re-parsed after each source code mutation. Therefore the initial request is expected to be significantly slower than the rest. With the current implementation of CodeProber, changing project configuration requires restarting CodeProber with new parameters. This means that the initial parse will likely happen without benefits from JVM optimization. We have benchmarked the full parse in two scenarios. Once where the parsing code had reached steady state, and once where the parsing code had just been loaded into the JVM ("Startup"). Full parse is measured for all project configurations **P**, but only with **N=1**, since multiple full parses wouldn't benefit from AST reuse, so there is no point in e.g **N=5**. The time for the initial parse can be seen in Table 1.

### 5.5 Overall Results

The case studies show that property probes are useful for a variety of analysis tools.

The performance measurements show that the overhead is negligible for practical use. J. Nielsen defined three time limits to keep in mind when talking about responsiveness [14]. According to that definition, responding in less than 0.1 second is enough to appear instant, and responding in less than 1 second is good enough to not interrupt the user's flow of thought. All normal actions in the benchmarks are faster than 0.1 second. The initial parse, which only happens once, is faster than 1 second even for the largest project (97 kLOC). In the smaller project configurations, the time usage consists almost entirely of probe administration. For the larger configurations we tested, a majority of the time is spent parsing-related tasks. This includes both reparsing modified files, and flushing away cached values from the reused parts of the AST. Flushing scales linearly with the size of the AST, which is why total time grows linearly too. With some sort of incremental flushing, we could possibly get better results. Either way, with current scaling characteristics we predict that performance is going to be acceptable even for much larger projects than the ones we tested.

Overall, property probes have shown to be very helpful in exploring how an analysis works, and for implementing and fixing features. The approach fits very well for analysis tools that use on-demand evaluation for individual properties, like the JastAdd-based tools we have tried it on. However, we think the approach can be very useful also for tools that do up-front evaluation, as long as the results can be tied to an AST with source text locations. In this case, properties can be explored interactively, although the user will of course have to wait for a possibly lengthy reanalysis if the underlying source text is edited.

## 6 Related Work

CodeProber allows interactive exploration of properties based on the source code. Earlier tools for debugging and exploring attribute grammars include, for example, Noosa [17] and DrAST [13]. Noosa is a special-purpose interactive debugger for compilers implemented in the Eli attribute grammar system. It supports, e.g., visualization of the AST, display of attributes of the AST, linking between source text and AST, monitoring the stream of abstract events during attribute evaluation, and setting breakpoints relating to such events. In contrast to CodeProber it does not have any concept of probes that are updated after changes to the source text.

DrAST is an interactive tool for visualizing JastAdd ASTs and inspecting AST node properties. It introduces a filtering language to collapse subtrees in order to reduce the visual complexity of the AST, and to specify which attributes to show directly in the tree for certain node types, possibly

conditionally. Individual attributes can also be inspected, but there is no concept of probes that are updated after source text changes.

The concept of node locators has relations to *origin tracking* [19]. This is a set of techniques for identifying where a node came from after tree rewrites. This is useful, for example, when generating error messages for transformed trees. Origin tracking has also been integrated with attribute grammars with higher-order attributes [21] (HOAs), which might be useful for improving locations for HOAs used in our CodeProber.

Node locators also have connections to *edit scripts*. Edit scripts describe differences between two versions of a source file. This can be used to generate detailed program diffs, or track nodes across multiple versions of a source file. However, existing techniques, like GumTree [8], IJM [11], MTDIFF [6], are focused on detecting differences between two files, without any knowledge of the actual input sequence that transformed one file to another. Our node locators require that we have input information available while editing. This is a limitation, but it also makes the algorithm much simpler. It might be possible to derive input information using edit scripts, and thus make it easier to integrate property probes with, for example, the Language Server Protocol (LSP).

Property probes in CodeProber can be viewed as being on liveness level 3 out of 6 according to Tanimoto [18]. Probes are automatically updated when either the input source file or the analysis tool have been changed, but do not predict user actions.

Property probes can also be compared to *watch* expressions found in many debuggers. Watch expressions typically only run while a debugging session is running, and the expressions are evaluated in the context of the current debugging session state. Property probes, on the other hand, are always active, and are evaluated without any state (except the source file contents that were used to initially construct the AST).

Beller, Spruit, Spinellis and Zaidman found that "developers spend surprisingly little time in the debugger" [5], citing complexity of debuggers as a potential reason. Many developers they surveyed preferred using "print debugging" instead, despite its limitations. This indicates to us that there is a need to develop new ways of exploring/debugging programs. It might be easier to develop debugging tools for particular use cases, like for exploring partial program analysis results. This paper represents one such tool.

## 7 Conclusion

We have presented the concept of property probes, an interactive mechanism for exploring program analyses in terms of the source code.

To support probes to be updated after edits, we introduced node locators with three kinds of steps: Child, TAL, and FN, and illustrated how they are used to robustly map between source code and the nodes in the program representation, and to handle synthetic nodes that have no representation in the source code.

We have developed CodeProber to support property probes, and discussed its client-server architecture and implementation. To validate our work, we successfully applied CodeProber to a number of tools for different languages and analyses, all based on Reference Attribute Grammars. This initial testing has already showed the utility of the tool; We are now using the tool extensively in our own work on program analysis. We have also shown through experiments that the overhead of using probes is very small, even if the analyzed project is large, giving latencies in the interactive tool that are far below the recommended limit of 0.1 seconds.

In going forward, we plan on performing more user studies. In addition to continuing to use CodeProber in our compilers course, we plan to use CodeProber in a program analysis course where students develop different analyses on top of a small procedural language.

## Acknowledgments

This work was funded by ELLIIT – Excellence Center at Linköping-Lund on Information Technology.

We would like to thank Idriss Riouak for continuously testing and giving feedback during the development of CodeProber.

## References

[1] [n. d.]. Apache(tm) FOP - a print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter. — xmlgraphics.apache.org. https://xmlgraphics.apache.org/fop/. [Accessed 28-Jun-2022].

[2] [n. d.]. Clang Static Analyzer — clang-analyzer.llvm.org. https://clang-analyzer.llvm.org/. [Accessed 22-Jun-2022].

[3] [n. d.]. JastAdd API Docs — extendj.org. http://extendj.org/doc/. [Accessed 28-Jun-2022].

[4] [n. d.]. Monaco Editor — microsoft.github.io. https://microsoft.github.io/monaco-editor/. [Accessed 22-Jun-2022].

[5] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. 572–583.

[6] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–671.

[7] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (oct 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[8] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[9] Niklas Fors and Görel Hedin. 2015. A JastAdd implementation of Oberon-0. *Sci. Comput. Program.* 114 (2015), 74–84. https://doi.org/10.1016/j.scico.2015.02.002

[10] Niklas Fors and Görel Hedin. 2016. Bloqqi: Modular Feature-Based Block Diagram Programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Amsterdam, Netherlands) *(Onward! 2016)*. Association for Computing Machinery, New York, NY, USA, 57–73. https://doi.org/10.1145/2986012.2986026

[11] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 264–274.

[12] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[13] Joel Lindholm, Johan Thorsberg, and Görel Hedin. 2016. DrAST: an inspection tool for attributed syntax trees (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 176–180. http://dl.acm.org/citation.cfm?id=2997378

[14] Jakob Nielsen. 1993. Response times: the three important limits. *Usability Engineering* (1993).

[15] Rohan Padhye, Koushik Sen, and Paul N Hilfinger. 2019. Chocopy: A programming language for compilers courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 41–45.

[16] Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. 2021. A Precise Framework for Source-Level Control-Flow Analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–11.

[17] Anthony M. Sloane. 1999. Debugging Eli-Generated Compilers With Noosa. In *Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1575)*, Stefan Jähnichen (Ed.). Springer, 17–31. https://doi.org/10.1007/978-3-540-49051-7_2

[18] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*, Brian Burg, Adrian Kuhn, and Chris Parnin (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[19] Arie Van Deursen, Paul Klint, and Frank Tip. 1993. Origin tracking. *Journal of Symbolic Computation* 15, 5-6 (1993), 523–545.

[20] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, Richard L. Wexelblat (Ed.). ACM, 131–145.

[21] Kevin Williams and Eric Van Wyk. 2014. Origin Tracking in Attribute Grammars. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 282–301. https://doi.org/10.1007/978-3-319-11245-9_16